8 Building forms with Tag Helpers

This chapter covers

- Building forms easily with Tag Helpers
- Generating URLs with the Anchor Tag Helper
- Using Tag Helpers to add functionality to Razor

In chapter 7 you learned about Razor templates and how to use them to generate the views for your application. By mixing HTML and C#, you can create dynamic applications that can display different data based on the request, the logged-in user, or any other data you can access.

Displaying dynamic data is an important aspect of many web applications, but it's typically only half of the story. As well as displaying data to the user, you often need the user to be able to submit data *back* to your application. You can use data to customize the view, or to update the application model by saving it to a database, for example. For traditional web applications, this data is usually submitted using an HTML form.

In chapter 6 you learned about model binding, which is how you *accept* the data sent by a user in a request and convert it into C# objects that you can use in your Razor Pages. You also learned about validation, and how important it is to validate the data sent in a request. You used `DataAnnotations` attributes to define the rules

associated with your models, as well as other associated metadata like the display name for a property.

The final aspect we haven't yet looked at is how to *build* the HTML forms that users use to send this data in a request. Forms are one of the key ways users will interact with your application in the browser, so it's important they're both correctly defined for your application and also user friendly. ASP.NET Core provides a feature to achieve this, called *Tag Helpers*.

Tag Helpers are new to ASP.NET Core. They're additions to Razor syntax that you can use to customize the HTML generated in your templates. Tag Helpers can be added to an otherwise standard HTML element, such as an `<input>`, to customize its attributes based on your C# model, saving you from having to write boilerplate code. Tag Helpers can also be standalone elements and can be used to generate completely customized HTML.

> **NOTE** Remember, Razor, and therefore Tag Helpers, are for server-side HTML rendering. You can't use Tag Helpers directly in frontend frameworks like Angular or React.

If you've used the previous version of ASP.NET, then Tag Helpers may sound reminiscent of HTML Helpers, which could also be used to generate HTML based on your C# classes. Tag Helpers are the logical successor to HTML Helpers, as they provide a more streamlined syntax than the previous, C#-focused helpers. HTML Helpers are still available in ASP.NET Core, so if you're converting some old templates to ASP.NET Core, you can still use them in your templates, but I won't be covering them in this book.

In this chapter, you'll primarily learn how to use Tag Helpers when building forms. They simplify the process of generating correct element names and IDs so that model binding can occur seamlessly when the form is sent back to your application. To put them into context, you're going to carry on building the currency converter application that you've seen in previous chapters. You'll add the ability to submit currency exchange requests to it, validate the data, and redisplay errors on the form using Tag Helpers to do the legwork for you, as shown in figure 8.1.

As you develop the application, you'll meet the most common Tag Helpers you'll encounter when working with forms. You'll also see how you can use Tag Helpers to simplify other common tasks, such as generating links, conditionally displaying data in your application, and ensuring users see the latest version of an image file when they refresh their browser.

To start, I'll talk a little about why you need Tag Helpers when Razor can already generate any HTML you like by combining C# and HTML in a file.

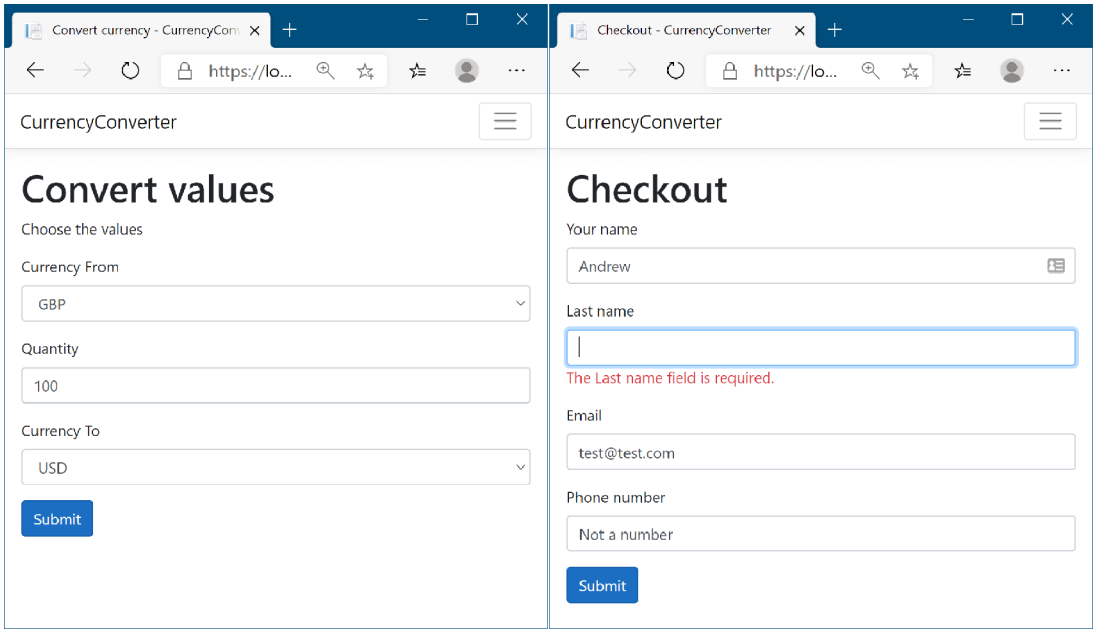

Figure 8.1 The currency converter application forms, built using Tag Helpers. The labels, drop-downs, input elements, and validation messages are all generated using Tag Helpers.

8.1 Catering to editors with Tag Helpers

One of the common complaints about the mixture of C# and HTML in Razor templates is that you can't easily use standard HTML editing tools with them; all the `@` and `{}` symbols in the C# code tend to confuse the editors. Reading the templates can be similarly difficult for people; switching paradigms between C# and HTML can be a bit jarring sometimes.

This arguably wasn't such a problem when Visual Studio was the only supported way to build ASP.NET websites, as it could obviously understand the templates without any issues and helpfully colorize the editor. But with ASP.NET Core going cross-platform, the desire to play nicely with other editors reared its head again.

This was one of the big motivations for Tag Helpers. They integrate seamlessly into the standard HTML syntax by adding what look to be attributes, typically starting with `asp-*`. They're most often used to generate HTML forms, as shown in the following listing. This listing shows a view from the first iteration of the currency converter application, in which you choose the currencies and quantity to convert.

Listing 8.1 User registration form using Tag Helpers

```
@page
@model ConvertModel
```

This is the view for the Razor Page Convert.cshtml. The Model type is ConvertModel.

```
<form method="post">
    <div class="form-group">
        <label asp-for="CurrencyFrom"></label>
        <input class="form-control" asp-for="CurrencyFrom" />
        <span asp-validation-for="CurrencyFrom"></span>
    </div>
    <div class="form-group">
        <label asp-for="Quantity"></label>
        <input class="form-control" asp-for="Quantity" />
        <span asp-validation-for="Quantity"></span>
    </div>
    <div class="form-group">
        <label asp-for="CurrencyTo"></label>
        <input class="form-control" asp-for="CurrencyTo" />
        <span asp-validation-for="CurrencyTo"></span>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

asp-for on Labels generates the caption for labels based on the view model.

asp-for on Inputs generates the correct type, value, name, and validation attributes for the model.

Validation messages are written to a span using Tag Helpers.

At first glance, you might not even spot the Tag Helpers, they blend in so well with the HTML! This makes it easy to edit the files with any standard HTML text editor. But don't be concerned that you've sacrificed readability in Visual Studio—as you can see in figure 8.2, elements with Tag Helpers are clearly distinguishable from the standard HTML `<div>` element and the standard HTML `class` attribute on the `<input>` element. The C# properties of the view model being referenced (`CurrencyFrom`, in this case) are also still shaded, as with other C# code in Razor files. And, of course, you get IntelliSense, as you'd expect.[1]

```
<form method="post">
    <div class="form-group">
        <label asp-for="Input.CurrencyFrom"></label>
        <input class="form-control" asp-for="Input.CurrencyFrom" />
        <span asp-validation-for="Input.CurrencyFrom" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Input.Qu"></label>
        <input class="form-con                .Quantity" />
        <span asp-validation-f                class="text-danger"></span>
    </div>
    <div class="form-group">
```

Quantity

Figure 8.2 In Visual Studio, Tag Helpers are distinguishable from normal elements by being bold and a different color, C# is shaded, and IntelliSense is available.

[1] Other editors like Visual Studio Code, JetBrains Rider, and Visual Studio for Mac also include syntax highlighting and IntelliSense support.

Tag Helpers are extra attributes on standard HTML elements (or new elements entirely) that work by modifying the HTML element they're attached to. They let you easily integrate your server-side values, such as those exposed on your `PageModel`, with the generated HTML.

Notice that listing 8.1 didn't specify the captions to display in the labels. Instead, you declaratively used `asp-for="CurrencyFrom"` to say, "for this `<label>`, use the `CurrencyFrom` property to work out what caption to use." Similarly, for the `<input>` elements, Tag Helpers are used to

- Automatically populate the value from the `PageModel` property.
- Choose the correct `id` and `name`, so that when the form is `POST`ed back to the Razor Page, the property will be model-bound correctly.
- Choose the correct input type to display (for example, a `number` input for the `Quantity` property).
- Display any validation errors, as shown in figure 8.3.[2]

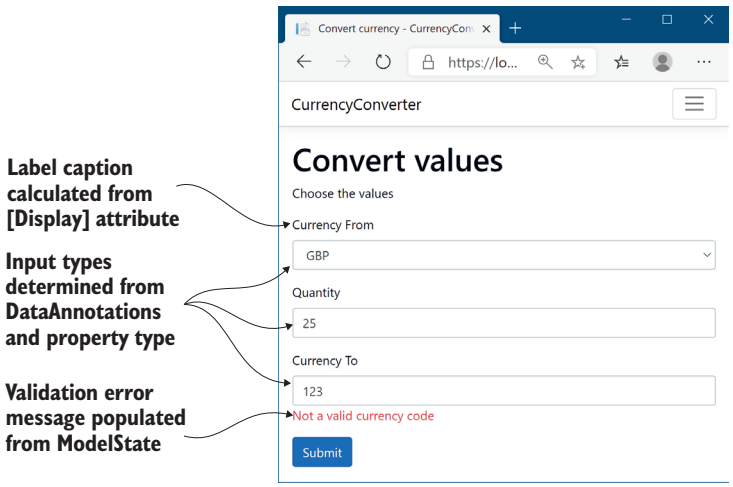

Figure 8.3 Tag Helpers hook into the metadata provided by `DataAnnotations` attributes, as well as the property types themselves. The Validation Tag Helper can even populate error messages based on the `ModelState`, as you saw in the last chapter on validation.

Tag Helpers can perform a variety of functions by modifying the HTML elements they're applied to. This chapter introduces a number of common Tag Helpers and how to use them, but it's not an exhaustive list. I don't cover all of the helpers that come out of the box in ASP.NET Core (there are more coming with every release!),

[2] To learn more about the internals of Tag Helpers, read the documentation at http://mng.bz/Idb0.

and you can easily create your own, as you'll see in chapter 19. Alternatively, you could use those published by others on NuGet or GitHub.[3] As with all of ASP.NET Core, Microsoft is developing Tag Helpers in the open on GitHub, so you can always take a look at the source code to see how they're implemented.

WebForms flashbacks

For those who used ASP.NET back in the day of WebForms, before the advent of the MVC pattern for web development, Tag Helpers may be triggering bad memories. Although the `asp-` prefix is somewhat reminiscent of ASP.NET Web Server control definitions, never fear—the two are different beasts.

Web Server controls were directly added to a page's backing C# class and had a broad scope that could modify seemingly unrelated parts of the page. Coupled with that, they had a complex lifecycle that was hard to understand and debug when things weren't working. The perils of trying to work with that level of complexity haven't been forgotten, and Tag Helpers aren't the same.

Tag Helpers don't have a lifecycle—they participate in the rendering of the element to which they're attached, and that's it. They can modify the HTML element they're attached to, but they can't modify anything else on your page, making them conceptually much simpler. An additional capability they bring is the ability to have multiple Tag Helpers acting on a single element—something Web Server controls couldn't easily achieve.

Overall, if you're writing Razor templates, you'll have a much more enjoyable experience if you embrace Tag Helpers as integral to its syntax. They bring a lot of benefits without obvious downsides, and your cross-platform-editor friends will thank you!

8.2 Creating forms using Tag Helpers

In this section you'll learn how to use some of the most useful Tag Helpers: Tag Helpers that work with forms. You'll learn how to use them to generate HTML markup based on properties of your `PageModel`, creating the correct `id` and `name` attributes and setting the `value` of the element to the model property's value (among other things). This capability significantly reduces the amount of markup you need to write manually.

Imagine you're building the checkout page for the currency converter application, and you need to capture the user's details on the checkout page. In chapter 6 you built a `UserBindingModel` model (shown in listing 8.2), added `DataAnnotations` attributes for validation, and saw how to model-bind it in a `POST` to a Razor Page. In this chapter you'll see how to create the view for it, by exposing the `UserBindingModel` as a property on your `PageModel`.

[3] A good example is Damian Edwards' (of the ASP.NET Core team) Tag Helper pack: https://github.com/DamianEdwards/TagHelperPack.

WARNING With Razor Pages, you often expose the same object in your view that you use for model binding. When you do this, you must be careful to not include sensitive values (that shouldn't be edited) in the binding model, to avoid mass-assignment attacks on your app.[4]

Listing 8.2 `UserBindingModel` for creating a user on a checkout page

```
public class UserBindingModel
{
    [Required]
    [StringLength(100, ErrorMessage = "Maximum length is {1}")]
    [Display(Name = "Your name")]
    public string FirstName { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "Maximum length is {1}")]
    [Display(Name = "Last name")]
    public string LastName { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Phone(ErrorMessage = "Not a valid phone number.")]
    [Display(Name = "Phone number")]
    public string PhoneNumber { get; set; }
}
```

The `UserBindingModel` is decorated with a number of `DataAnnotations` attributes. In chapter 6 you saw that these attributes are used during model validation when the model is bound to a request, before the page handler is executed. These attributes are *also* used by the Razor templating language to provide the metadata required to generate the correct HTML when you use Tag Helpers.

You can use the pattern I described in chapter 6, exposing a `UserBindindModel` as an `Input` property of your `PageModel`, to use the model for both model binding and in your Razor view:

```
public class CheckoutModel: PageModel
{
    [BindProperty]
    public UserBindingModel Input { get; set; }
}
```

With the help of the `UserBindingModel` property, Tag Helpers, and a little HTML, you can create a Razor view that lets the user enter their details, as shown in figure 8.4.

[4] You can read more about over-posting attacks on my blog at http://mng.bz/RXw0.

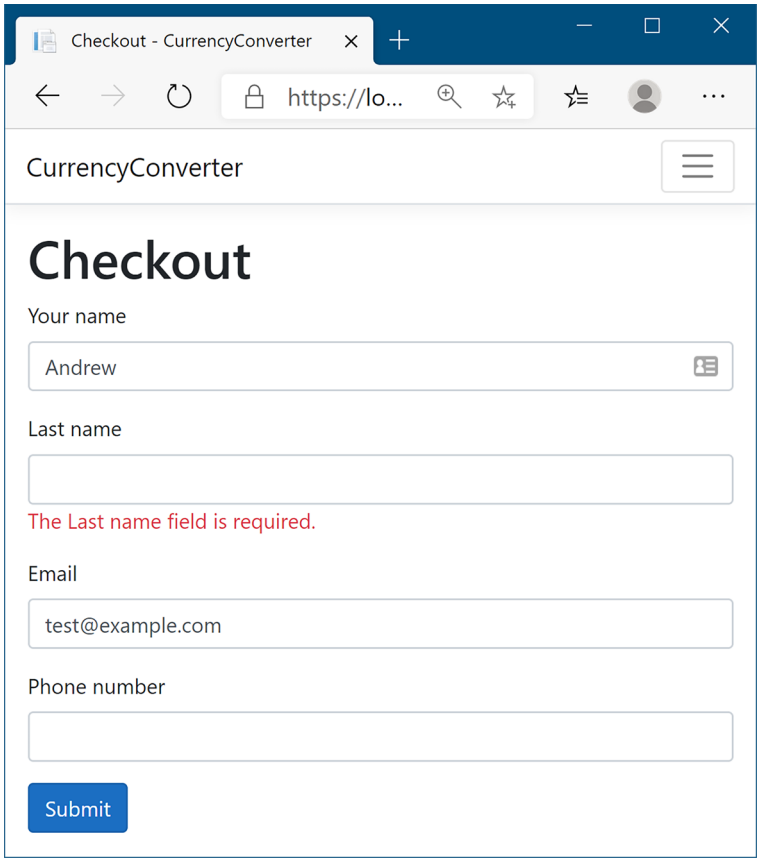

Figure 8.4 The checkout page for an application. The HTML is generated based on a `UserBindingModel`, using Tag Helpers to render the required element values, input types, and validation messages.

The Razor template to generate this page is shown in listing 8.3. This code uses a variety of tag helpers, including

- A Form Tag Helper on the `<form>` element
- Label Tag Helpers on the `<label>`
- Input Tag Helpers on the `<input>`
- Validation Message Tag Helpers on `<span>` validation elements for each property in the `UserBindingModel`

Listing 8.3 Razor template for binding to `UserBindingModel` on the checkout page

```
@page
@model CheckoutModel
@{
```

The CheckoutModel is the PageModel, which exposes a UserBindingModel on the Input property.

```
    ViewData["Title"] = "Checkout";
}
<h1>@ViewData["Title"]</h1>
<form asp-page="Checkout">
    <div class="form-group">
        <label asp-for="Input.FirstName"></label>
        <input class="form-control" asp-for="Input.FirstName" />
        <span asp-validation-for="Input.FirstName"></span>
    </div>
    <div class="form-group">
        <label asp-for="Input.LastName"></label>
        <input class="form-control" asp-for="Input.LastName" />
        <span asp-validation-for="Input.LastName"></span>
    </div>
    <div class="form-group">
        <label asp-for="Input.Email"></label>
        <input class="form-control" asp-for="Input.Email" />
        <span asp-validation-for="Input.Email"></span>
    </div>
    <div class="form-group">
        <label asp-for="Input.PhoneNumber"></label>
        <input class="form-control" asp-for="Input.PhoneNumber" />
        <span asp-validation-for="Input.PhoneNumber"></span>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Form Tag Helpers use routing to determine the URL the form will be posted to.

The Label Tag Helper uses DataAnnotations on a property to determine the caption to display.

The Input Tag Helper uses DataAnnotations to determine the type of input to generate.

The Validation Tag Helper displays error messages associated with the given property.

You can see the HTML markup that this template produces in listing 8.4. This Razor markup and the resulting HTML produces the results you saw in figure 8.4. You can see that each of the HTML elements with a Tag Helper has been customized in the output: the `<form>` element has an `action` attribute, the `<input>` elements have an `id` and `name` based on the name of the referenced property, and both the `<input>` and `<span>` have `data-*` attributes for validation.

Listing 8.4 HTML generated by the Razor template on the checkout page

```
<form action="/Checkout" method="post">
  <div class="form-group">
    <label for="Input_FirstName">Your name</label>
    <input class="form-control" type="text"
      data-val="true" data-val-length="Maximum length is 100"
      id="Input_FirstName" data-val-length-max="100"
      data-val-required="The Your name field is required."
      Maxlength="100" name="Input.FirstName" value="" />
    <span data-valmsg-for="Input.FirstName"
      class="field-validation-valid" data-valmsg-replace="true"></span>
  </div>
  <div class="form-group">
    <label for="Input_LastName">Your name</label>
    <input class="form-control" type="text"
      data-val="true" data-val-length="Maximum length is 100"
      id="Input_LastName" data-val-length-max="100"
      data-val-required="The Your name field is required."
```

```
      Maxlength="100" name="Input.LastName" value="" />
    <span data-valmsg-for="Input.LastName"
      class="field-validation-valid" data-valmsg-replace="true"></span>
  </div>
  <div class="form-group">
    <label for="Input_Email">Email</label>
    <input class="form-control" type="email" data-val="true"
      data-val-email="The Email field is not a valid e-mail address."
      Data-val-required="The Email field is required."
      Id="Input_Email" name="Input.Email" value="" />
    <span class="text-danger field-validation-valid"
      data-valmsg-for="Input.Email" data-valmsg-replace="true"></span>
    </div>
  <div class="form-group">
    <label for="Input_PhoneNumber">Phone number</label>
    <input class="form-control" type="tel" data-val="true"
      data-val-phone="Not a valid phone number." Id="Input_PhoneNumber"
      name="Input.PhoneNumber" value="" />
    <span data-valmsg-for="Input.PhoneNumber"
      class="text-danger field-validation-valid"
      data-valmsg-replace="true"></span>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
  <input name="__RequestVerificationToken" type="hidden"
    value="CfDJ8PkYhAINFx1JmYUVIDWbpPyy_TRUNCATED" />
</form>
```

Wow, that's a lot of markup! If you're new to working with HTML, this might all seem a little overwhelming, but the important thing to notice is that *you didn't have to write most of it*! The Tag Helpers took care of most of the plumbing for you. That's basically Tag Helpers in a nutshell; they simplify the fiddly mechanics of building HTML forms, leaving you to concentrate on the overall design of your application instead of writing boilerplate markup.

> **NOTE** If you're using Razor to build your views, Tag Helpers will make your life easier, but they're entirely optional. You're free to write raw HTML without them, or to use the legacy HTML Helpers.

Tag Helpers simplify and abstract the process of HTML generation, but they generally try to do so without getting in your way. If you need the final generated HTML to have a particular attribute, you can add it to your markup. You can see that in the previous listings where `class` attributes are defined on `<input>` elements, such as `<input class="form-control" asp-for="Input.FirstName" />`. They pass untouched from Razor to the HTML output.

> **TIP** This is different from the way HTML Helpers worked in the previous version of ASP.NET; HTML helpers often require jumping through hoops to set attributes in the generated markup.

Even better than this, you can also set attributes that are normally generated by a Tag Helper, like the `type` attribute on an `<input>` element. For example, if the

FavoriteColor property on your PageModel was a string, then by default Tag Helpers would generate an <input> element with type="text". Updating your markup to use the HTML5 color picker type is trivial; set the type explicitly in your Razor view:

```
<input type="color" asp-for="FavoriteColor" />
```

> **TIP** HTML5 adds a huge number of features, including lots of form elements that you may not have come across before, such as range inputs and color pickers. We're not going to cover them in this book, but you can read about them on the Mozilla Developer Network website at http://mng.bz/qOc1.

In this section, you'll build the currency calculator Razor templates from scratch, adding Tag Helpers as you find you need them. You'll probably find you use most of the common form Tag Helpers in every application you build, even if it's on a simple login page.

8.2.1 The Form Tag Helper

The first thing you need to start building your HTML form is, unsurprisingly, the <form> element. In the previous example, the <form> element was augmented with an asp-page Tag Helper attribute:

```
<form asp-page="Checkout">
```

This results in action and method attributes being added to the final HTML, indicating which URL the form should be sent to when it's submitted and the HTTP verb to use:

```
<form action="/Checkout" method="post">
```

Setting the asp-page attribute allows you to specify a different Razor Page in your application that the form will be posted to when it's submitted. If you omit the asp-page attribute, the form will post back to the same URL it was served from. This is *very* common with Razor Pages. You normally handle the result of a form post in the same Razor Page that is used to display it.

> **WARNING** If you omit the asp-page attribute, you must manually add the method="post" attribute. It's important to add this attribute so the form is sent using the POST verb, instead of the default GET verb. Using GET for forms can be a security risk.

The asp-page attribute is added by a FormTagHelper. This Tag Helper uses the value provided to generate a URL for the action attribute, using the URL generation features of routing that I described at the end of chapter 5.

> **NOTE** Tag Helpers can make multiple attributes available on an element. Think of them like properties on a Tag Helper configuration object. Adding a single asp- attribute activates the Tag Helper on the element. Adding additional attributes lets you override further default values of its implementation.

The Form Tag Helper makes several other attributes available on the `<form>` element that you can use to customize the generated URL. Hopefully you'll remember from chapter 5 that you can set route values when generating URLs. For example, if you have a Razor Page called Product.cshtml that uses the directive

```
@page "{id}"
```

then the full route template for the page would be `"Product/{id}"`. To correctly generate the URL for this page, you must provide the `{id}` route value. How can you set that value using the Form Tag Helper?

The Form Tag Helper defines an `asp-route-*` wildcard attribute that you can use to set arbitrary route parameters. Set the `*` in the attribute to the route parameter name. For example, to set the `id` route parameter, you'd set the `asp-route-id` value (it's shown with a fixed value of `5` in the following example, but it more commonly would be dynamic):

```
<form asp-page="Product" asp-route-id="5">
```

Based on the route template of the Product.cshtml Razor Page, this would generate the following markup:

```
<form action="/Product/5" method="post">
```

You can add as many `asp-route-*` attributes as necessary to your `<form>` to generate the correct `action` URL. You can also set the Razor Page handler to use the `asp-page-handler` attribute. This ensures the form `POST` will be handled by the handler you specify.

> **NOTE** The Form Tag Helper has many additional attributes, such as `asp-action` and `asp-controller`, that you generally won't need to use with Razor Pages. Those are only useful if you're using MVC controllers with views. In particular, look out for the `asp-route` attribute—this is *not* the same as the `asp-route-*` attribute. The former is used to specify a *named* route (not used with Razor Pages), and the latter is used to specify the route *values* to use during URL generation.

Just as for all other Razor constructs, you can use C# values from your `PageModel` (or C# in general) in Tag Helpers. For example, if the `ProductId` property of your `PageModel` contains the value required for the `{id}` route value, you could use

```
<form asp-page="Product" asp-route-id="@Model.ProductId">
```

The main job of the Form Tag Helper is to generate the `action` attribute, but it performs one additional, important function: generating a hidden `<input>` field needed to prevent *cross-site request forgery* (CSRF) attacks.

DEFINITION *Cross-site request forgery* (CSRF) attacks are a website exploit that can allow actions to be executed on your website by an unrelated malicious website. You'll learn about them in detail in chapter 18.

You can see the generated hidden `<input>` at the bottom of the `<form>` in listing 8.4; it's named `__RequestVerificationToken` and contains a seemingly random string of characters. This field won't protect you on its own, but I'll describe in chapter 18 how it's used to protect your website. The Form Tag Helper generates it by default, so you generally won't need to worry about it, but if you need to disable it, you can do so by adding `asp-antiforgery="false"` to your `<form>` element.

The Form Tag Helper is obviously useful for generating the `action` URL, but it's time to move on to more interesting elements—those that you can see in your browser!

8.2.2 The Label Tag Helper

Every `<input>` field in your currency converter application needs to have an associated label so the user knows what the `<input>` is for. You could easily create those yourself, manually typing the name of the field and setting the `for` attribute as appropriate, but luckily there's a Tag Helper to do that for you.

The Label Tag Helper is used to generate the caption (the visible text) and the `for` attribute for a `<label>` element, based on the properties in the `PageModel`. It's used by providing the name of the property in the `asp-for` attribute:

```
<label asp-for="FirstName"></label>
```

The Label Tag Helper uses the `[Display]` `DataAnnotations` attribute that you saw in chapter 6 to determine the appropriate value to display. If the property you're generating a label for doesn't have a `[Display]` attribute, the Label Tag Helper will use the name of the property instead. Consider this model in which the `FirstName` property has a `[Display]` attribute, but the `Email` property doesn't:

```
public class UserModel
{
    [Display(Name = "Your name")]
    public string FirstName { get; set; }
    public string Email { get; set; }
}
```

The following Razor

```
<label asp-for="FirstName"></label>
<label asp-for="Email"></label>
```

would generate this HTML:

```
<label for="FirstName">Your name</label>
<label for="Email">Email</label>
```

The caption text inside the `<label>` element uses the value set in the `[Display]` attribute, or the property name in the case of the `Email` property. Also note that the `for` attribute has been generated with the name of the property. This is a key bonus of using Tag Helpers—it hooks in with the element IDs generated by other Tag Helpers, as you'll see shortly.

> **NOTE** The `for` attribute is important for accessibility. It specifies the ID of the element to which the label refers. This is important for users who are using a screen reader, for example, as they can tell what property a form field relates to.

As well as properties on the `PageModel`, you can also reference sub-properties on child objects. For example, as I described in chapter 6, it's common to create a nested class in a Razor Page, expose that as a property, and decorate it with the `[BindProperty]` attribute:

```
public class CheckoutModel: PageModel
{
    [BindProperty]
    public UserBindingModel Input { get; set; }
}
```

You can reference the `FirstName` property of the `UserBindingModel` by "dotting" into the property as you would in any other C# code. Listing 8.3 shows more examples of this.

```
<label asp-for="Input.FirstName"></label>
<label asp-for="Input.Email"></label>
```

As is typical with Tag Helpers, the Label Tag Helper won't override values that you set yourself. If, for example, you don't want to use the caption generated by the helper, you could insert your own manually. The following code,

```
<label asp-for="Email">Please enter your Email</label>
```

would generate this HTML:

```
<label for="Email">Please enter your Email</label>
```

As ever, you'll generally have an easier time with maintenance if you stick to the standard conventions and don't override values like this, but the option is there. Next up is a biggie: the Input and Textarea Tag Helpers.

8.2.3 The Input and Textarea Tag Helpers

Now you're getting into the meat of your form—the `<input>` elements that handle user input. Given that there's such a wide array of possible input types, there's a variety of different ways they can be displayed in the browser. For example, Boolean values are typically represented by a `checkbox` type `<input>` element, whereas integer values

would use a `number` type `<input>` element, and a date would use the `date` type, as shown in figure 8.5.

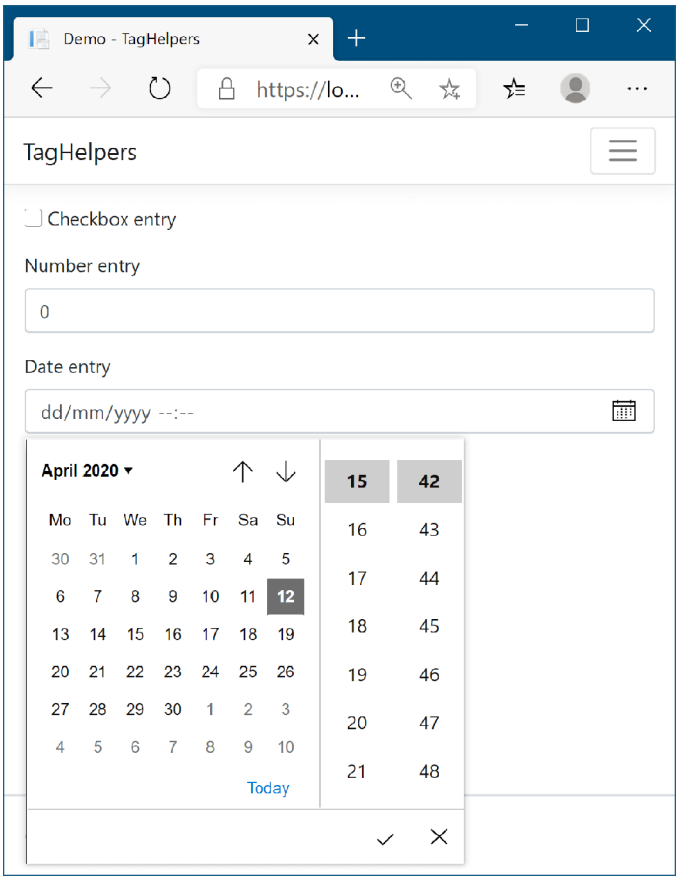

Figure 8.5 Various input element types. The exact way in which each type is displayed varies by browser.

To handle this diversity, the Input Tag Helper is one of the most powerful Tag Helpers. It uses information based on both the type of the property (`bool`, `string`, `int`, and so on) and any `DataAnnotations` attributes applied to it (`[EmailAddress]` and `[Phone]`, among others) to determine the type of the `input` element to generate. The `DataAnnotations` are also used to add `data-val-*` client-side validation attributes to the generated HTML.

Consider the `Email` property from listing 8.2 that was decorated with the `[EmailAddress]` attribute. Adding an `<input>` is as simple as using the `asp-for` attribute:

```
<input asp-for="Input.Email" />
```

The property is a `string`, so ordinarily the Input Tag Helper would generate an `<input>` with `type="text"`. But the addition of the `[EmailAddress]` attribute provides additional metadata about the property. Consequently, the Tag Helper generates an HTML5 `<input>` with `type="email"`:

```
<input type="email" id="Input_Email" name="Input.Email"
    value="test@example.com" data-val="true"
    data-val-email="The Email Address field is not a valid e-mail address."
    Data-val-required="The Email Address field is required."
    />
```

You can take a whole host of things away from this example. First, the `id` and `name` attributes of the HTML element have been generated from the name of the property. The value of the `id` attribute matches the value generated by the Label Tag Helper in its `for` attribute, `Input_Email`. The value of the `name` attribute preserves the "dot" notation, `Input.Email`, so that model binding works correctly when the field is `POST`ed to the Razor Page.

Also, the initial `value` of the field has been set to the value currently stored in the property (`"test@example.com"`, in this case). The `type` of the element has also been set to the HTML5 `email` type, instead of using the default `text` type.

Perhaps the most striking addition is the swath of `data-val-*` attributes. These can be used by client-side JavaScript libraries such as jQuery to provide client-side validation of your `DataAnnotations` constraints. Client-side validation provides instant feedback to users when the values they enter are invalid, providing a smoother user experience than can be achieved with server-side validation alone, as I described in chapter 6.

Client-side validation

In order to enable client-side validation in your application, you need to add some jQuery libraries to your HTML pages. In particular, you need to include the jQuery, jQuery-validation, and jQuery-validation-unobtrusive JavaScript libraries. You can do this in a number of ways, but the simplest is to include the script files at the bottom of your view using

```
<script src="~/lib/jquery-
            validation/dist/jquery.validate.min.js"></script>
<script src="~/lib/jquery-validation-
            unobtrusive/jquery.validate.unobtrusive.min.js"></script>
```

The default templates include these scripts for you, in a handy partial template that you can add to your page in a `Scripts` section. If you're using the default layout and need to add client-side validation to your view, add the following section somewhere on your view:

```
@section Scripts{
    @Html.Partial("_ValidationScriptsPartial")
}
```

This partial view references files in your wwwroot folder. The default _layout template includes jQuery itself, as that's required by the front-end component library Bootstrap.[a]

[a] You can also load these files from a content delivery network (CDN). If you wish to take this approach, you should consider scenarios where the CDN is unavailable or compromised, as I discuss in this blog post: http://mng.bz/2e6d.

The Input Tag Helper tries to pick the most appropriate template for a given property based on `DataAnnotations` attributes or the type of the property. Whether this generates the exact `<input>` type you need may depend, to an extent, on your application. As always, you can override the generated `type` by adding your own `type` attribute to the element in your Razor template. Table 8.1 shows how some of the common data types are mapped to `<input>` types, and how the data types themselves can be specified.

Table 8.1 Common data types, how to specify them, and the input element type they map to

Data type	How it's specified	Input element type
`byte, int, short, long, uint`	Property type	`number`
`decimal, double, float`	Property type	`text`
`bool`	Property type	`checkbox`
`string`	Property type, `[DataType(DataType.Text)]` attribute	`text`
`HiddenInput`	`[HiddenInput]` attribute	`hidden`
`Password`	`[Password]` attribute	`password`
`Phone`	`[Phone]` attribute	`tel`
`EmailAddress`	`[EmailAddress]` attribute	`email`
`Url`	`[Url]` attribute	`url`
`Date`	`DateTime` property type, `[DataType(DataType.Date)]` attribute	`date`

The Input Tag Helper has one additional attribute that can be used to customize the way data is displayed: `asp-format`. HTML forms are entirely string-based, so when the `value` of an `<input>` is set, the Input Tag Helper must take the value stored in the property and convert it to a `string`. Under the covers, this performs a `string.Format()` on the property's value, passing in the format string.

The Input Tag Helper uses a default format string for each different data type, but with the `asp-format` attribute, you can set the specific format string to use. For example,

you could ensure a `decimal` property, `Dec`, is formatted to three decimal places with the following code:

```
<input asp-for="Dec" asp-format="{0:0.000}" />
```

If the `Dec` property had a value of 1.2, this would generate HTML similar to

```
<input type="text" id="Dec" name="Dec" value="1.200">
```

> **NOTE** You may be surprised that `decimal` and `double` types are rendered as `text` fields and not as `number` fields. This is due to several technical reasons, predominantly related to the way some cultures render numbers with commas and spaces. Rendering as text avoids errors that would only appear in certain browser-culture combinations.

In addition to the Input Tag Helper, ASP.NET Core provides the Textarea Tag Helper. This works in a similar way, using the `asp-for` attribute, but it's attached to a `<textarea>` element instead:

```
<textarea asp-for="BigtextValue"></textarea>
```

This generates HTML similar to the following. Note that the property value is rendered inside the tag, and `data-val-*` validation elements are attached as usual:

```
<textarea data-val="true" id="BigtextValue" name="BigtextValue"
    data-val-length="Maximum length 200." data-val-length-max="200"
    data-val-required="The Multiline field is required." >This is some text,
I'm going to display it
in a text area</textarea>
```

Hopefully, this section has hammered home how much typing Tag Helpers can cut down on, especially when using them in conjunction with `DataAnnotations` for generating validation attributes. But this is more than reducing the number of keystrokes required; Tag Helpers ensure that the markup generated is *correct*, and has the correct `name`, `id`, and format to automatically bind your binding models when they're sent to the server.

With `<form>`, `<label>`, and `<input>` under your belt, you're able to build most of your currency converter forms. Before we look at displaying validation messages, there's one more element to look at: the `<select>`, or drop-down, input.

8.2.4 The Select Tag Helper

As well as `<input>` fields, a common element you'll see on web forms is the `<select>` element, or drop-downs and list boxes. Your currency converter application, for example, could use a `<select>` element to let you pick which currency to convert from a list.

By default, this element shows a list of items and lets you select one, but there are several variations, as shown in figure 8.6. As well as the normal drop-down box, you could show a list box, add multiselection, or display your list items in groups.

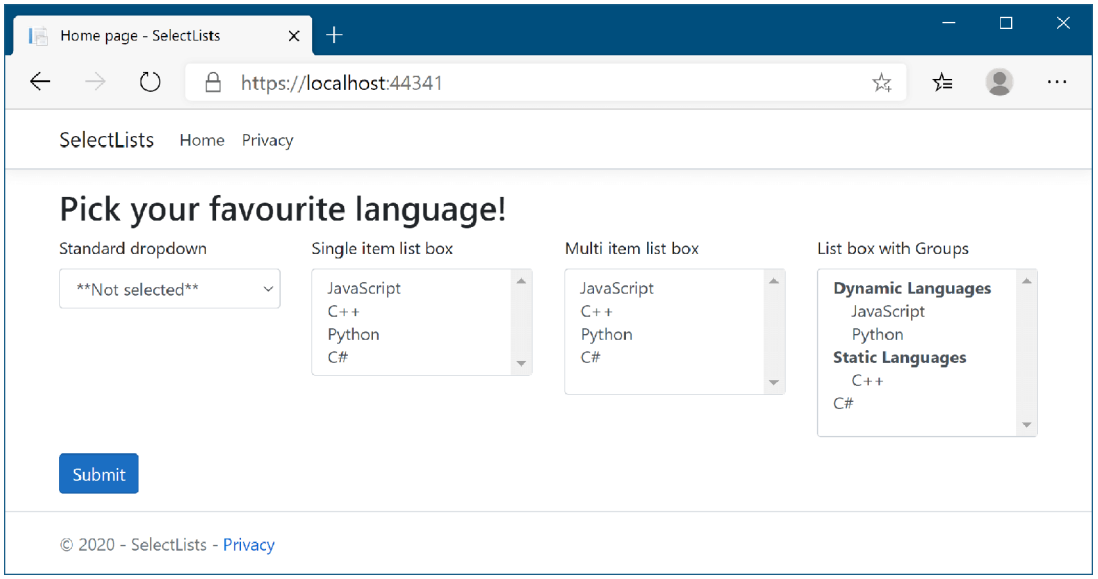

Figure 8.6 Some of the many ways to display `<select>` elements using the Select Tag Helper.

To use `<select>` elements in your Razor code, you'll need to include two properties in your `PageModel`: one property for the list of options to display, and one to hold the value (or values) selected. For example, listing 8.5 shows the properties on the `PageModel` used to create the three left-most select lists shown in figure 8.6. Displaying groups requires a slightly different setup, as you'll see shortly.

Listing 8.5 View model for displaying select element drop-downs and list boxes

```
public class SelectListsModel: PageModel
{
    [BindProperty]
    public class InputModel Input { get; set; }

    public IEnumerable<SelectListItem> Items { get; set; }
        = new List<SelectListItem>
    {
        new SelectListItem{Value= "csharp", Text="C#"},
        new SelectListItem{Value= "python", Text= "Python"},
        new SelectListItem{Value= "cpp", Text="C++"},
        new SelectListItem{Value= "java", Text="Java"},
        new SelectListItem{Value= "js", Text="JavaScript"},
        new SelectListItem{Value= "ruby", Text="Ruby"},
    };


    public class InputModel
    {
```

The InputModel for binding the user's selections to the select boxes

The list of items to display in the select boxes

```
        public string SelectedValue1 { get; set; }
        public string SelectedValue2 { get; set; }
        public IEnumerable<string> MultiValues { get; set; }
    }
}
```

These properties will hold the values selected by the single-selection select boxes.

To create a multiselect list box, use an IEnumerable<>.

This listing demonstrates a number of aspects of working with `<select>` lists:

- `SelectedValue1/SelectedValue2`—Used to hold the value selected by the user. They're model-bound to the value selected from the drop-down/listbox and used to preselect the correct item when rendering the form.
- `MultiValues`—Used to hold the selected values for a multiselect list. It's an `IEnumerable`, so it can hold more than one selection per `<select>` element.
- `Items`—Provides the list of options to display in the `<select>` elements. Note that the element type must be `SelectListItem`, which exposes the `Value` and `Text` properties, to work with the Select Tag Helper. This isn't part of the `InputModel`, as we don't want to model-bind these items to the request—they would normally be loaded directly from the application model or hardcoded.

NOTE The Select Tag Helper only works with `SelectListItem` elements. That means you'll normally have to convert from an application-specific list set of items (for example, a `List<string>` or `List<MyClass>`) to the UI-centric `List<SelectListItem>`.

The Select Tag Helper exposes the `asp-for` and `asp-items` attributes that you can add to `<select>` elements. As for the Input Tag Helper, the `asp-for` attribute specifies the property in your `PageModel` to bind to. The `asp-items` attribute is provided for the `IEnumerable<SelectListItem>` to display the available `<option>` elements.

TIP It's common to want to display a list of `enum` options in a `<select>` list. This is so common that ASP.NET Core ships with a helper for generating a `SelectListItem` for any `enum`. If you have an `enum` of the `TEnum` type, you can generate the available options in your view using `asp-items="Html.GetEnumSelectList<TEnum>()"`.

The following listing shows how to display a drop-down list, a single-selection list box, and a multiselection list box. It uses the `PageModel` from the previous listing, binding each `<select>` list to a different property, but reusing the same `Items` list for all of them.

Listing 8.6 Razor template to display a select element in three different ways

```
@page
@model SelectListsModel
<select asp-for="Input.SelectedValue1"
    asp-items="Model.Items"></select>
<select asp-for="Input.SelectedValue2"
    asp-items="Model.Items" size="4"></select>
```

Creates a standard drop-down select list by binding to a standard property in asp-for

Creates a single-select list box of height 4 by providing the standard HTML size attribute

```
<select asp-for="Input.MultiValues"
    asp-items="Model.Items"></select>
```

Creates a multiselect list box by binding to an IEnumerable property in asp-for

Hopefully, you can see that the Razor for generating a drop-down `<select>` list is almost identical to the Razor for generating a multiselect `<select>` list. The Select Tag Helper takes care of adding the `multiple` HTML attribute to the generated output if the property it's binding to is an `IEnumerable`.

> **WARNING** The `asp-for` attribute *must not* include the `Model.` prefix. The `asp-items` attribute, on the other hand, *must* include it if referencing a property on the `PageModel`. The `asp-items` attribute can also reference other C# items, such as objects stored in `ViewData`, but using a `PageModel` property is the best approach.

You've seen how to bind three different types of select list so far, but the one I haven't yet covered from figure 8.6 is how to display groups in your list boxes using `<optgroup>` elements. Luckily, nothing needs to change in your Razor code; you just have to update how you define your `SelectListItems`.

The `SelectListItem` object defines a `Group` property that specifies the `SelectListGroup` the item belongs to. The following listing shows how you could create two groups and assign each list item to either a "dynamic" or "static" group, using a `PageModel` similar to that shown in listing 8.5. The final list item, `C#`, isn't assigned to a group, so it will be displayed as normal, without an `<optgroup>`.

Listing 8.7 Adding `Groups` to `SelectListItems` to create `optgroup` elements

```
public class SelectListsModel: PageModel
{
    [BindProperty]
    public IEnumerable<string> SelectedValues { get; set; }
    public IEnumerable<SelectListItem> Items { get; set; }

    public SelectListsModel()
    {
        var dynamic = new SelectListGroup { Name = "Dynamic" };
        var stat = new SelectListGroup { Name = "Static" };
        Items = new List<SelectListItem>
        {
            new SelectListItem {
                Value= "js",
                Text="Javascript",
                Group = dynamic
            },
            new SelectListItem {
                Value= "cpp",
                Text="C++",
                Group = stat
            },
            new SelectListItem {
```

Used to hold the selected values where multiple selections are allowed

Initializes the list items in the constructor

Creates a single instance of each group to pass to SelectListItems

Sets the appropriate group for each SelectListItem

```
                Value= "python",
                Text="Python",
                Group = dynamic
            },
            new SelectListItem {
                Value= "csharp",
                Text="C#",
            }
        };
    }
}
```

Sets the appropriate group for each SelectListItem

If a SelectListItem doesn't have a Group, it won't be added to an <optgroup>.

With this in place, the Select Tag Helper will generate `<optgroup>` elements as necessary when rendering the Razor to HTML. This Razor template,

```
@page
@model SelectListsModel
<select asp-for="SelectedValues" asp-items="Model.Items"></select>
```

would be rendered to HTML as follows:

```
<select id="SelectedValues" name="SelectedValues" multiple="multiple">
    <optgroup label="Dynamic">
        <option value="js">JavaScript</option>
        <option value="python">Python</option>
    </optgroup>
    <optgroup label="Static Languages">
        <option value="cpp">C++</option>
    </optgroup>
    <option value="csharp">C#</option>
</select>
```

Another common requirement when working with `<select>` elements is to include an option in the list that indicates that no value has been selected, as shown in figure 8.7. Without this extra option, the default `<select>` drop-down will always have a value, and it will default to the first item in the list.

You can achieve this in one of two ways: you could either add the "not selected" option to the available `SelectListItems`, or you could manually add the option to the Razor, for example by using

```
<select asp-for="SelectedValue" asp-items="Model.Items">
    <option Value = "">**Not selected**</option>
</select>
```

This will add an extra `<option>` at the top of your `<select>` element, with a blank `Value` attribute, allowing you to provide a "no selection" option for the user.

TIP Adding a "no selection" option to a `<select>` element is so common that you might want to create a partial view to encapsulate this logic.

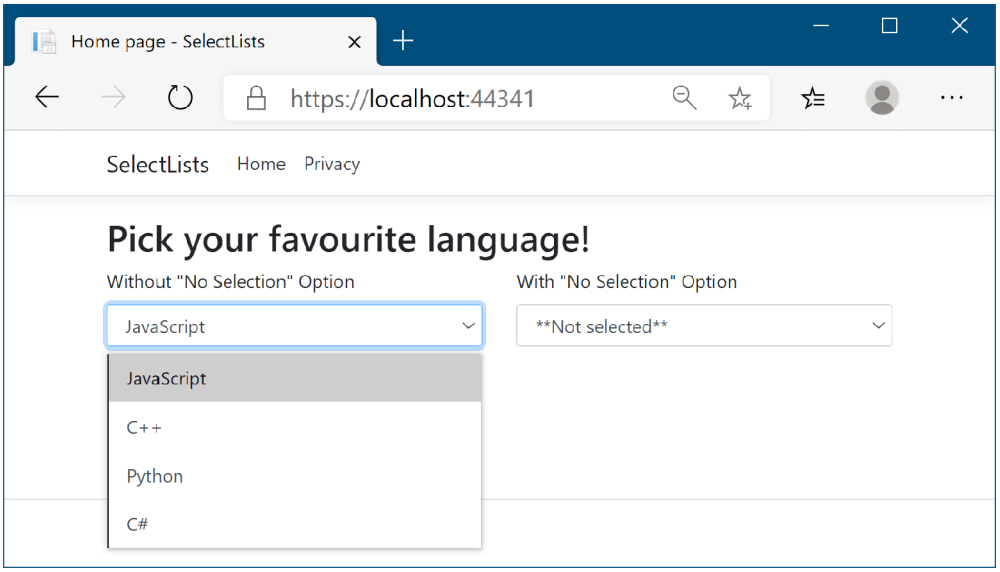

Figure 8.7 Without a "not selected" option, the `<select>` element will always have a value. This may not be the behavior you desire if you don't want an `<option>` to be selected by default.

With the Input Tag Helper and Select Tag Helper under your belt, you should be able to create most of the forms that you'll need. You have all the pieces you need to create the currency converter application now, with one exception.

Remember, whenever you accept input from a user, you should always validate the data. The Validation Tag Helpers provide a way to display model validation errors to the user on your form, without having to write a lot of boilerplate markup.

8.2.5 The Validation Message and Validation Summary Tag Helpers

In section 8.2.3 you saw that the Input Tag Helper generates the necessary `data-val-*` validation attributes on form input elements themselves. But you also need somewhere to display the validation messages. This can be achieved for each property in your view model using the Validation Message Tag Helper applied to a `<span>` by using the `asp-validation-for` attribute:

```
<span asp-validation-for="Email"></span>
```

When an error occurs during client-side validation, the appropriate error message for the referenced property will be displayed in the `<span>`, as shown in figure 8.8. This `<span>` element will also be used to show appropriate validation messages if server-side validation fails and the form is being redisplayed.

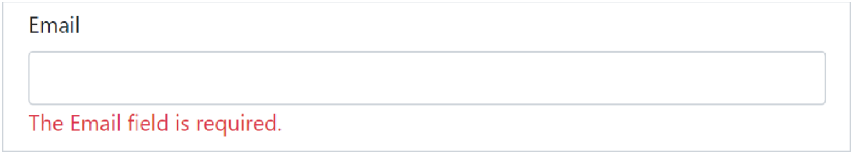

Figure 8.8 Validation messages can be shown in an associated `<span>` by using the Validation Message Tag Helper.

Any errors associated with the `Email` property stored in `ModelState` will be rendered in the element body, and the element will have appropriate attributes to hook into jQuery validation:

```
<span class="field-validation-valid" data-valmsg-for="Email"
  data-valmsg-replace="true">The Email Address field is required.</span>
```

The validation error shown in the element will be replaced when the user updates the `Email` `<input>` field and client-side validation is performed.

> **NOTE** For further details on `ModelState` and server-side model validation, see chapter 6.

As well as displaying validation messages for individual properties, you can also display a summary of all the validation messages in a `<div>` by using the Validation Summary Tag Helper, shown in figure 8.9. This renders a `<ul>` containing a list of the `ModelState` errors.

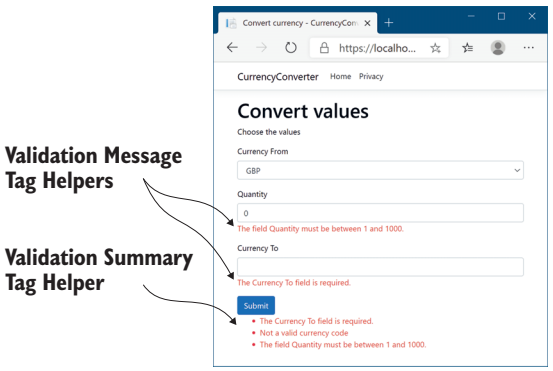

Figure 8.9 Form showing validation errors. The Validation Message Tag Helper is applied to `<span>`, close to the associated input. The Validation Summary Tag Helper is applied to a `<div>`, normally at the top or bottom of the form.

The Validation Summary Tag Helper is applied to a `<div>` using the `asp-validation-summary` attribute and providing a `ValidationSummary` `enum` value, such as

```
<div asp-validation-summary="All"></div>
```

The `ValidationSummary` enum controls which values are displayed, and it has three possible values:

- `None`—Don't display a summary. (I don't know why you'd use this.)
- `ModelOnly`—Only display errors that are *not* associated with a property.
- `All`—Display errors either associated with a property or with the model.

The Validation Summary Tag Helper is particularly useful if you have errors associated with your page that aren't specific to a single property. These can be added to the model state by using a blank key, as shown in listing 8.8. In this example, the property validation passed, but we provide additional model-level validation to check that we aren't trying to convert a currency to itself.

Listing 8.8 Adding model-level validation errors to the `ModelState`

```
public class ConvertModel : PageModel
{
    [BindProperty]
    public InputModel Input { get; set; }

    [HttpPost]
    public IActionResult OnPost()
    {
        if(Input.CurrencyFrom == Input.CurrencyTo)
        {
            ModelState.AddModelError(
                string.Empty,
                "Cannot convert currency to itself");
        }
        if (!ModelState.IsValid)
        {
            return Page();
        }

        //store the valid values somewhere etc
        return RedirectToPage("Checkout");
    }
}
```

Can't convert currency to itself

Adds model-level error, not tied to a specific property, by using empty key

If there are any property-level or model-level errors, display them.

Without the Validation Summary Tag Helper, the model-level error would still be added if the user used the same currency twice, and the form would be redisplayed. Unfortunately, there would have been no visual cue to the user indicating why the form did not submit—obviously that's a problem! By adding the Validation Summary Tag Helper, the model-level errors are shown to the user so they can correct the problem, as shown in figure 8.10.

NOTE For simplicity, I added the validation check to the page handler. A better approach might be to create a custom validation attribute to achieve this instead. That way, your handler stays lean and sticks to the single responsibility principle. You'll see how to achieve this in chapter 20.

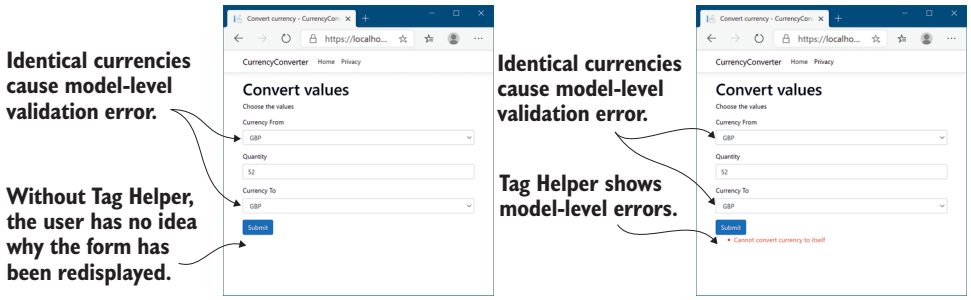

Figure 8.10 Model-level errors are only displayed by the Validation Summary Tag Helper. Without one, users won't have any indication that there were errors on the form, and so won't be able to correct them.

This section has covered most of the common Tag Helpers available for working with forms, including all the pieces you need to build the currency converter forms. They should give you everything you need to get started building forms in your own applications. But forms aren't the only area in which Tag Helpers are useful; they're generally applicable any time you need to mix server-side logic with HTML generation.

One such example is generating links to other pages in your application using routing-based URL generation. Given that routing is designed to be fluid as you refactor your application, keeping track of the exact URLs the links should point to would be a bit of a maintenance nightmare if you had to do it by hand. As you might expect, there's a Tag Helper for that: the Anchor Tag Helper.

8.3 Generating links with the Anchor Tag Helper

At the end of chapter 5, I showed how you could generate URLs for links to other pages in your application from inside your page handlers by using `ActionResults`. Views are the other common place where you need to link to other pages in your application, normally by way of an `<a>` element with an `href` attribute pointing to the appropriate URL.

In this section I'll show how you can use the Anchor Tag Helper to generate the URL for a given Razor Page using routing. Conceptually, this is almost identical to the way the Form Tag Helper generates the `action` URL, as you saw in section 8.2.1. For the most part, using the Anchor Tag Helper is identical too; you provide `asp-page` and `asp-page-handler` attributes, along with `asp-route-*` attributes as necessary. The default Razor Page templates use the Anchor Tag Helper to generate the links shown in the navigation bar using the code in the following listing.

Listing 8.9 Using the Anchor Tag Helper to generate URLs in _Layout.cshtml

```
<ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
```

```
        <a class="nav-link text-dark"
            asp-area="" asp-page="/Index">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-area="" asp-page="/Privacy">Privacy</a>
    </li>
</ul>
```

As you can see, each `<a>` element has an `asp-page` attribute. This Tag Helper uses the routing system to generate an appropriate URL for the `<a>`, resulting in the following markup:

```
<ul class="nav navbar-nav">
    <li class="nav-item">
        <a class="nav-link text-dark" href="/">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" href="/Privacy">Privacy</a>
    </li>t
</ul>
```

The URLs use default values where possible, so the `Index` Razor Page generates the simple `"/"` URL instead of `"/Index"`.

If you need more control over the URL generated, the Anchor Tag Helper exposes several additional properties you can set, which will be used during URL generation. These are the most commonly used with Razor Pages:

- `asp-page`—Sets the Razor Page to execute.
- `asp-page-handler`—Sets the Razor Page handler to execute.
- `asp-area`—Sets the area route parameter to use. Areas can be used to provide an additional layer of organization to your application.[5]
- `asp-host`—If set, the link will point to the provided host and will generate an absolute URL instead of a relative URL.
- `asp-protocol`—Sets whether to generate an http or https link. If set, it will generate an absolute URL instead of a relative URL.
- `asp-route-*`—Sets the route parameters to use during generation. Can be added multiple times for different route parameters.

By using the Anchor Tag Helper and its attributes, you generate your URLs using the routing system, as described in chapter 5. This reduces the duplication in your code by removing the hardcoded URLs you'd otherwise need to embed in all your views.

If you find yourself writing repetitive code in your markup, chances are someone has written a Tag Helper to help with it. The Append Version Tag Helper in the

[5] I won't cover areas in detail in this book. They're an optional aspect of MVC that are often only used on large projects. You can read about them here: http://mng.bz/3X64.

following section is a great example of using Tag Helpers to reduce the amount of fiddly code required.

8.4 Cache-busting with the Append Version Tag Helper

A common problem with web development, both when developing and when an application goes into production, is ensuring that browsers are all using the latest files. For performance reasons, browsers often cache files locally and reuse them for subsequent requests, rather than calling your application every time a file is requested.

Normally this is great—most of the static assets in your site rarely change, so caching them significantly reduces the burden on your server. Think of an image of your company logo—how often does that change? If every page shows your logo, then caching the image in the browser makes a lot of sense.

But what happens if it *does* change? You want to make sure users get the updated assets as soon as they're available. A more critical requirement might be if the JavaScript files associated with your site change. If users end up using cached versions of your JavaScript, they might see strange errors, or your application might appear broken to them.

This conundrum is a common one in web development, and one of the most common ways for handling it is to use a cache-busting query string.

> **DEFINITION** A *cache-busting query string* adds a query parameter to a URL, such as `?v=1`. Browsers will cache the response and use it for subsequent requests to the URL. When the resource changes, the query string is also changed, for example to `?v=2`. Browsers will see this as a request for a new resource and will make a fresh request.

The biggest problem with this approach is that it requires you to update a URL every time an image, CSS, or JavaScript file changes. This is a manual step that requires updating every place the resource is referenced, so it's inevitable that mistakes are made. Tag Helpers to the rescue! When you add a `<script>`, `<img>`, or `<link>` element to your application, you can use Tag Helpers to automatically generate a cache-busting query string:

```
<script src="~/js/site.js" asp-append-version="true"></script>
```

The `asp-append-version` attribute will load the file being referenced and generate a unique hash based on its contents. This is then appended as a unique query string to the resource URL:

```
<script src="/js/site.js?v=EWaMeWsJBYWmL2g_KkgXZQ5nPe"></script>
```

As this value is a hash of the file contents, it will remain unchanged as long as the file isn't modified, so the file will be cached in users' browsers. But if the file is modified, the hash of the contents will change and so will the query string. This ensures browsers

are always served the most up-to-date files for your application without your having to worry about manually updating every URL whenever you change a file.

So far in this chapter you've seen how to use Tag Helpers for forms, link generation, and cache busting. You can also use Tag Helpers to conditionally render different markup depending on the current environment. This uses a technique you haven't seen yet, where the Tag Helper is declared as a completely separate element.

8.5 Using conditional markup with the Environment Tag Helper

In many cases, you want to render different HTML in your Razor templates depending on whether your website is running in a development or production environment. For example, in development you typically want your JavaScript and CSS assets to be verbose and easy to read, but in production you'd process these files to make them as small as possible. Another example might be the desire to apply a banner to the application when it's running in a testing environment, which is removed when you move to production, as shown in figure 8.11.

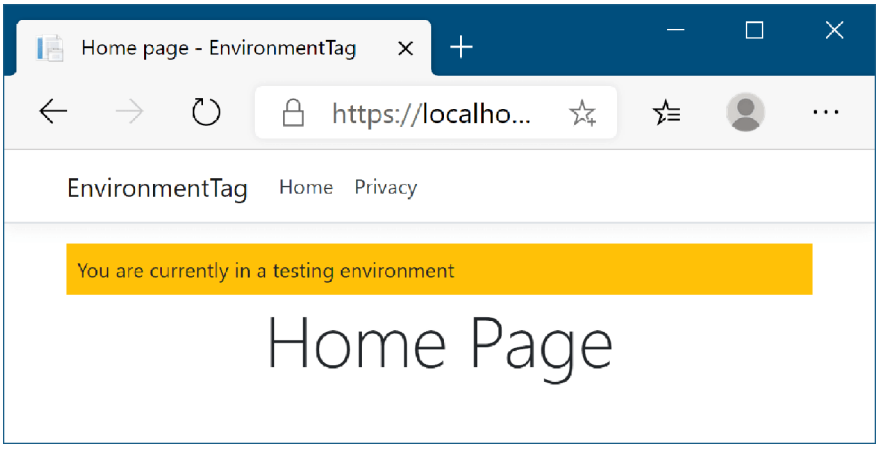

Figure 8.11 The warning banner will be shown whenever you're running in a testing environment, to make it easy to distinguish from production.

> **NOTE** You'll learn about configuring your application for multiple environments in chapter 11.

You've already seen how to use C# to add `if` statements to your markup, so it would be perfectly possible to use this technique to add an extra `div` to your markup when the current environment has a given value. If we assume that the `env` variable contains the current environment, you could use something like this:

```
@if(env == "Testing" || env == "Staging")
{
    <div class="warning">You are currently on a testing environment</div>
}
```

There's nothing wrong with this, but a better approach would be to use the Tag Helper paradigm to keep your markup clean and easy to read. Luckily, ASP.NET Core comes with the `EnvironmentTagHelper`, which can be used to achieve the same result in a slightly clearer way:

```
<environment include="Testing,Staging">
    <div class="warning">You are currently on a testing environment</div>
</environment>
```

This Tag Helper is a little different from the others you've seen before. Instead of augmenting an existing HTML element using an `asp-` attribute, the *whole element* is the Tag Helper. This Tag Helper is completely responsible for generating the markup, and it uses an attribute to configure it.

Functionally, this Tag Helper is identical to the C# markup (although for now I've glossed over how the `env` variable could be found), but it's more declarative in its function than the C# alternative. You're obviously free to use either approach, but personally I like the HTML-like nature of Tag Helpers.

We've reached the end of this chapter on Tag Helpers, and with it, our first look at building traditional web applications that display HTML to users. In the last part of the book, we'll revisit Razor templates, and you'll learn how to build custom components like custom Tag Helpers and view components. For now, you have everything you need to build complex Razor layouts—the custom components can help tidy up your code down the line.

Part 1 of this book has been a whistle-stop tour of how to build Razor Page applications with ASP.NET Core. You now have the basic building blocks to start making simple ASP.NET Core applications. In the second part of this book, I'll show you some of the additional features you'll need to understand to build complete applications. But before we get to that, I'll take a chapter to discuss building Web APIs.

I've mentioned the Web API approach previously, in which your application serves data using the MVC framework, but instead of returning user-friendly HTML, it returns machine-friendly JSON. In the next chapter you'll see why and how to build a Web API, take a look at an alternative routing system designed for APIs, and learn how to generate JSON responses to requests.

Summary

- With Tag Helpers, you can bind your data model to HTML elements, making it easier to generate dynamic HTML while remaining editor friendly.
- As with Razor in general, Tag Helpers are for server-side rendering of HTML only. You can't use them directly in frontend frameworks, such as Angular or React.

- Tag Helpers can be standalone elements or can attach to existing HTML using attributes. This lets you both customize HTML elements and add entirely new elements.
- Tag Helpers can customize the elements they're attached to, add additional attributes, and customize how they're rendered to HTML. This can greatly reduce the amount of markup you need to write.
- Tag Helpers can expose multiple attributes on a single element. This makes it easier to configure the Tag Helper, as you can set multiple, separate values.
- You can add the `asp-page` and `asp-page-handler` attributes to the `<form>` element to set the `action` URL using the URL generation feature of Razor Pages.
- You specify route values to use during routing with the Form Tag Helper using `asp-route-*` attributes. These values are used to build the final URL or are passed as query data.
- The Form Tag Helper also generates a hidden field that you can use to prevent CSRF attacks. This is added automatically and is an important security measure.
- You can attach the Label Tag Helper to a `<label>` using `asp-for`. It generates an appropriate `for` attribute and caption based on the `[Display] Data-Annotations` attribute and the `PageModel` property name.
- The Input Tag Helper sets the `type` attribute of an `<input>` element to the appropriate value based on a bound property's `Type` and any `DataAnnotations` applied to it. It also generates the `data-val-*` attributes required for client-side validation. This significantly reduces the amount of HTML code you need to write.
- To enable client-side validation, you must add the necessary JavaScript files to your view for jQuery validation and unobtrusive validation.
- The Select Tag Helper can generate drop-down `<select>` elements as well as list boxes, using the `asp-for` and `asp-items` attributes. To generate a multiselect `<select>` element, bind the element to an `IEnumerable` property on the view model. You can use these approaches to generate several different styles of select box.
- The items supplied in `asp-for` must be an `IEnumerable<SelectListItem>`. If you try to bind another type, you'll get a compile-time error in your Razor view.
- You can generate an `IEnumerable<SelectListItem>` for an `enum TEnum` using the `Html.GetEnumSelectList<TEnum>()` helper method. This saves you having to write the mapping code yourself.
- The Select Tag Helper will generate `<optgroup>` elements if the items supplied in `asp-for` have an associated `SelectListGroup` on the `Group` property. Groups can be used to separate items in select lists.
- Any extra additional `<option>` elements added to the Razor markup will be passed through to the final HTML. You can use these additional elements to easily add a "no selection" option to the `<select>` element.

- The Validation Message Tag Helper is used to render the client- and server-side validation error messages for a given property. This gives important feedback to your users when elements have errors. Use the `asp-validation-for` attribute to attach the Validation Message Tag Helper to a `<span>`.
- The Validation Summary Tag Helper is used to display validation errors for the model, as well as for individual properties. You can use model-level properties to display additional validation that doesn't apply to just one property. Use the `asp-validation-summary` attribute to attach the Validation Summary Tag Helper to a `<div>`.
- You can generate `<a>` URLs using the Anchor Tag Helper. This helper uses routing to generate the `href` URL using `asp-page`, `asp-page-handler`, and `asp-route-*` attributes, giving you the full power of routing.
- You can add the `asp-append-version` attribute to `<link>`, `<script>`, and `<img>` elements to provide cache-busting capabilities based on the file's contents. This ensures users cache files for performance reasons, yet still always get the latest version of files.
- You can use the Environment Tag Helper to conditionally render different HTML based on the app's current execution environment. You can use this to render completely different HTML in different environments if you wish.